Journal of Computational and Applied Mathematics 30 (1990) 283–294
North-Holland

A multicolour SOR method for the finite-element method

C.H. WU
Numerical Analysis Group, Department of Mathematics, Trinity College, Dublin 2, Ireland

Received 23 December 1988
Revised 5 December 1989

Abstract: In this paper we present several algorithms to reorder unknowns in a finite-element mesh so that we can use the multicolour SOR method to solve the corresponding linear system on a pipelined computer or on a parallel computer. We also discuss the assembling process by reordering elements with our algorithms. Numerical tests on a pipelined computer indicate the efficiency of the multicolour SOR method.

Keywords: Multicolour SOR method, parallel computing, finite-element method.

1. Introduction

We consider the solution of a sparse $n \times n$ linear system of equations

$$Ax = b \tag{1.1}$$

arising from the finite-element discretisation of partial differential equations; in particular we discuss the method of Successive Overrelaxation (SOR) on vector or parallel computers.

Several authors, e.g., Adams et al. [2,3], Hayes [7] and Lambiotte [9], have observed that for the finite-difference discretisation the classical Red/Black ordering of the grid points may increase the speed of the SOR method on vector or parallel computers. In the paper of Kincaid et al. [8] ITPACK is used to solve sparse linear systems by a variety of iterative methods on CYBER 205 and on CRAY-1. Their results show that the Red/Black ordered SOR method decreases the CPU-time considerably, about three to five times, compared with the natural rowwise ordered SOR method for their test problem (see Section 3, Test 2).

While the Red/Black ordering allows an efficient implementation of the SOR method for the five-point difference scheme, it does not work for higher-order finite-difference or finite-element discretisation or for more general elliptic equations which contain mixed partial derivative terms. In order to apply this Red/Black ordering to more general cases, for example, for higher-order finite-difference discretisations, Adams et al. [2,3] generalised this to multicoloured ordering. The word "multicolour" comes from the fact that we can use more than two colours to partition grid points into several sets. In each set, the grid points are decoupled from each other. We identify the grid points in each set with one colour. This kind of SOR method using the multicolour ordering is called the multicolour SOR method.

Using the concept of data flow, Adams and Jordan [1] showed that for several finite-difference stencils SOR is colour-blind, that is, the rate of convergence of the SOR method remains the same for any order of colour chosen. They also showed that the multicolour SOR method retains the same rate of convergence as the natural rowwise SOR method for a wide range of meshes for the discretisation of a partial differential equation.

For the finite-element method, George [6] introduced the so-called nested dissection method to order grid points. His purpose is to get a matrix which has a form more suitable for the direct method. For example, for the LDL^T factorisation it decreases the number of arithmetic operations and the storage requirement considerably. Only recently has work intensified on ordering grid points on a general finite-element mesh for iterative methods.

Berger et al. [5] use multicolour orderings for the assembly of finite-element equations. Their purpose is to avoid data conflict, i.e., to avoid reading and writing the data in one memory unit at the same time. We shall explain this in detail in Section 2.2.

In this paper, we are concerned with the problem of ordering grid points in a finite-element mesh for the SOR method and ordering elements for the assembling process. The purpose of our ordering is to obtain a linear system of equations which is more suitable for computations on vector or parallel computers. In Section 2, we begin by analysing the graph of the global matrix of the finite-element method and present our algorithms for vector computers and parallel computers. Secondly, we deal with the assembly of finite-element equations. Using the same idea as that for ordering grid points, we present an algorithm to assemble elements in parallel. In Section 3, we show some numerical tests for the multicolour SOR method on a vector computer. In Section 4, we give our conclusions.

2. The multicolour algorithms

2.1. The algorithms

It is well known that the Jacobi iteration is a "perfect" parallel method since the unknowns are decoupled during each iteration. This means that during each Jacobian iteration we could get the new value of each unknown at the same time if there were enough processors. But the Jacobi iterative method converges too slowly to be used in practice. Other kinds of iterative methods, such as the Gauss–Seidel iterative method and the SOR method, converge faster than the Jacobi iterative method but they do not have this decoupling property for each unknown. For this kind of method (in this paper we only consider the SOR method) and for the sparse finite-element matrix, we want to find a way of ordering the unknowns so that the reordered unknowns have a local decoupling property, i.e., the unknowns can be separated into several groups and in each group they are decoupled. Reordering unknowns permutes the matrix A in (1.1) into a matrix B of the form

$$B = \begin{bmatrix} D_1 & B_{12} & \cdots & \cdots & B_{1p} \\ B_{21} & D_2 & \ddots & \cdots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & B_{p-1,p} \\ B_{p1} & \cdots & \cdots & B_{p,p-1} & D_p \end{bmatrix}, \tag{2.1}$$

where the D_i are diagonal matrices, p is the number of groups and the B_{ij} are sparse matrices. With this local decoupling property the Gauss–Seidel iterative method and the SOR method can be carried out in the same fashion as the Jacobi iteration by using p vectors corresponding to each group of unknowns.

Therefore we should concentrate on the matrix B in (2.1) at first. In the graph of B (for the graph of a matrix, cf. [11, pp.19,20, pp.97–131]), the nodes are separated into several groups. In each colour group there are no edges between the nodes. For a finite-element mesh, if two grid points are within one element, there might be an edge between the corresponding nodes in the graph of the global matrix.

We separate grid points in a finite-element mesh by a process called colouring. We colour the grid points so that no two grid points in an element have the same colour. This ensures that all grid points of a given colour are decoupled from one another.

Keeping this in mind, we are ready to give our first algorithm for a general mesh of the finite-element method. We shall assume that A is symmetric, this assumption may be easily removed later.

Algorithm C1

(1) Give an arbitrary ordering of the grid points.

(2) Find the maximum degree of the grid points in the mesh and denote it $d_{\max}$. Set the number of colours $NC = d_{\max} + 1$. (The degree of a point is the number of lines incident with the point.)

(3) Set the colour number of the first point P_1, $C_1 = 1$.

(4) Suppose the first k points have the colour numbers C_i, $1 \leqslant C_i \leqslant NC$, $1 \leqslant i \leqslant k$. For the $(k+1)$th point P_{k+1}, using the graph of the global matrix or the finite-element mesh, find points in the set $\{P_1, P_2, \ldots, P_k\}$ which are connected with P_{k+1}, say these points are $\{P_{i_1}, P_{i_2}, \ldots, P_{i_k}\}$ which correspond to a set of colour numbers, say S_k, $S_k = \{C_{j_1}, C_{j_2}, \ldots, C_{j_l}\}$.

(5) Choose the colour number for P_{k+1}. We choose the first number in the set $\{1, 2, \ldots, NC\} \setminus S_k$ as C_{k+1}.

(6) If $k+1 < n$, go to step (4), else continue.

(7) Examine the set $\{C_1, C_2, \ldots, C_n\}$ to find the real number of colours used which is also denoted as NC.

(8) Find the points belonging to each colour group and reorder them according to the colour number order, i.e., order points belonging to the first colour group first, then second colour second, and so on, in each colour group just keep to the original order.

In the end we reorder all the grid points in the finite-element mesh and in each colour group the points are not connected with each other. The global matrix will have the form of (2.1).

Notice that from step (4) to step (6) in Algorithm C1, we give every grid point a colour number C_i. This process will not take up a lot of CPU-time, since generally for a finite-element mesh the degree of every point is much less than the number of grid points. So step (8) uses most of the available colouring time. (We call this process colouring the grid points.) In step (8), we have to search each unknown several times. Of course, we could make an improvement at this point, by simply increasing the storage requirements. We can use different arrays to store node numbers of different colour groups. In this way, we can omit step (8) of Algorithm C1. Since we

do not know exactly how many points belong to each colour group before they are coloured, we have no way of estimating the increase in storage requirement.

Analysing this algorithm, we notice that the grid points mainly belong to the first groups of points. We can also see this from the numerical tests in Section 3. So, the ordering produced by this algorithm is more suitable to the vector or pipeline computer. Since on the vector computer, the longer the vector of unknowns that can be calculated at the same time, the shorter the CPU-time will be.

For parallel computers or array computers, it would be more efficient to have nearly the same number of unknowns in each group if we assign each processor to deal with one colour group. We notice that the reason that Algorithm C1 failed at this point is that it depends on the colour number order, i.e., colour 1 has more opportunity to be chosen than the others, colour 2 has more opportunity than the others except colour 1, and so on. So if we change slightly step (5) of Algorithm C1 to give each colour the same opportunity to be chosen, we can expect the number of nodes in different colour groups to be much closer to each other. So we give the Algorithm C2.

Algorithm C2

(1), (2), (3), (4) are the same as those in Algorithm C1.

(5) We choose the colour number for P_{k+1} in the set $\{1, 2, \ldots, NC\} \setminus S_k$ at random. We can do this by letting number $1, 2, \ldots, NC$ permute all the time, and always choose the first one in that permuted colour number set except S_k.

(6) If $k + 1 < n$, go to step (4), else continue.

(7) The same as step (8) in Algorithm C1.

Since in this case all the colours should be chosen, it is not necessary to keep step (7) of Algorithm C1 in Algorithm C2.

We notice that in Algorithm C2 we also choose the number of processors as NC if more than $d_{\max} + 1$ processors are available. So if we change step (2) of Algorithm C2 to

(2) Find $d_{\max}$. If the number of processors NP is bigger than $d_{\max} + 1$, then let NC equal NP, we have an algorithm which we call *Algorithm C2'*. This can be used by a parallel computer more efficiently.

Sometimes we may have fewer processors than $d_{\max} + 1$. We may confine the number of colours to the number of processors. In this case we cannot promise the permuted matrix B will still be of the form (2.1), but we can still obtain a matrix of the following form:

$$B = \begin{pmatrix} D_1 & \cdots & B_{1,NC} & B_{1,NC+1} \\ \vdots & \ddots & \vdots & \vdots \\ B_{NC,1} & \cdots & D_{NC} & B_{NC,NC+1} \\ B_{NC+1,1} & \cdots & B_{NC+1,NC} & B_{NC+1,NC+1} \end{pmatrix},$$

where $D_1, D_2, \ldots, D_{NC}$ are diagonal matrices, B_{ij} are general sparse matrices.

Now the algorithm is like this.

Algorithm C3

(1) Give an arbitrary ordering of the mesh points.

(2) Find $d_{\max}$. If the number of processors NP is smaller than $d_{\max} + 1$, then let $NC = NP - 1$.

(3) Set the colour number of the first grid point P_1, $C_1 = 1$.
(4) Find the colour number of grid points $P_{i_1}, P_{i_2}, \ldots, P_{i_k}$ as in Algorithm C1 to get S_k.
(5) When $\{1, 2, \ldots, NC\} \setminus S_k = \emptyset$, set $C_{k+1} = NC + 1$; otherwise do the same as in step (5) of Algorithm C2.
(6) If $k + 1 < n$, go to step (4), else continue.
(7) Reorder the mesh points as in step (8) of Algorithm C1.

2.2. The assembly problem

We can also use this idea to assemble element matrices into the global matrix in parallel. The assembly is just updating the global matrix by adding in the element matrix. When we use an array computer or MIMD system, we can compute every element matrix concurrently on different processors. But since two or more updating tasks running on different processors will possibly perform load-add-store operations on identical elements of A, an order of elements must be defined in order to avoid this conflict, which we call data conflict.

Berger et al. [5] presented an algorithm similar to Algorithm C1 and they also gave an algorithm to keep the balance of the number of elements for each colour group.

Berger's algorithm

(1) Reorder elements in the mesh according to their decreasing degree (i.e., the number of other elements connected with it). We denote the newly ordered list of elements as $\{p_i\}$, $i = 1, \ldots, n$. Let $j = 1$.
(2) Take element $p_{j_1} = p_j$ and find all elements p_{j_k} in the ordered list such that p_{j_k} is not adjacent to any p_{j_l}, $l = 1, \ldots, k - 1$. This forms a partition S_j.
(3) Suppress all elements of S_j from the list. If the list is empty, halt the process, otherwise let j be the new first element and iterate step (2).

Berger et al. balanced the number of elements in each colour by shifting part of the first group (say S_1) to the last group S_n, and so on. We need to judge whether the number of elements in each group is equal to n/p or not (p is the number of processors) and to decide whether to shift S_2 to S_n or part of S_1 to $S_n - 1$.

From the above Berger's algorithm we can see that the first two steps will use a lot of CPU-time, since we need to go through the whole element list several times. And the way they balance the number of elements in each colour group is expensive. With the above algorithms of node colouring, we can change our point of view in the following way to give another method of colouring elements. We choose one point in every element and connect the points in different elements which have a common edge (in two dimensions). With these points and edges we obtain a new mesh, we call it a dual mesh of the original one. Now we can use our Algorithm C2 on this dual mesh and get another kind of assembly. Since every grid point in the dual mesh is corresponding to an element in the original mesh, colouring grid points in the dual mesh is equivalent to ordering elements in the original mesh.

To colour elements, we can actually use an array corresponding to the nodes to store the colour numbers of each element. In this way, we can avoid going through the whole element list every time we colour an element. For example, we consider a triangular mesh in two dimensions. For the first element we give it colour number 1 and we put this information in the three vertices

P_1, P_2, P_3, i.e., for each node we have a vector C_{P_i}, which records the colour number of the elements with the ith grid point as their vertex; then $C_{P_1}(1) = 1$, $C_{P_2}(1) = 1$, $C_{P_3}(1) = 1$. For the second element we find it has the vertices P_2, P_3, P_4; we go through C_{P_2} and C_{P_3} and we find colour number 1 has already been used; then we can give the second element colour number 2 and put this information into its vertices P_2, P_3, P_4, i.e., $C_{P_2}(2) = 2$, $C_{P_3}(2) = 2$, $C_{P_4}(1) = 2$. Keeping on this process we can give each element a colour number. This process has a high parallel property and further study is necessary.

2.3. *Implementation of the algorithms*

For the finite-element method, we can put all these colouring algorithms in the preprocess part, that is, before we start to do the real calculation, we choose a suitable way to order the grid nodes in the mesh so that the global matrix will have a suitable form (such as (2.1)). We can do this by first getting the graph or pattern of the global matrix. If the pattern of matrix A is unsymmetric we can use our algorithms for the matrix $A + A^{\mathrm{T}}$, to reorder A. Similarly we can get an order of elements for the assembly.

3. Numerical results

To test the efficiency of our algorithms, we use two tests, one is the finite-difference method for the Laplace equation in two dimensions, the other is the finite-element method for a test problem used by Kincaid et al. [8]. We do our tests on the FPS M64/30, a pipelined computer. In all the tests for the SOR method the initial iterate is $u^{(0)} = 0$ and the stopping criterion is

$$\max_{1 \leqslant i \leqslant n} \left| u_i^{(k+1)} - u_i^{(k)} \right| < 5.0 \cdot 10^{-6},$$

where $u_i^{(k)}$ is the ith value of u in the kth iteration.

Test 1. We use the five-point finite-difference stencil for the two-dimensional Laplace equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

on a uniform $n \times n$ grid, in a rectangular region Ω, subject to the Dirichlet boundary condition. This is the so-called model problem.

We consider four kinds of ordering of the unknowns, the natural rowwise ordering, ordering the unknowns by the C1 algorithm, which is exactly the Red/Black ordering, ordering the unknowns by the C2 algorithm using 3 colours (we call it 3-colour ordering) and ordering by the C2 algorithm using 5 colours (we call it 5-colour ordering). For $n = 10$, the number of grid points in each colour group is 50 for the Red/Black ordering, 33 to 34 for the 3-colour ordering and 20 for the 5-colour ordering. The patterns of the matrices for these four kinds of ordering are shown in Figs. 1(a)–1(d), where dots indicate nonzero entries.

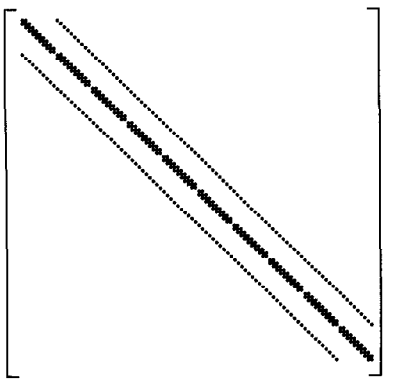


Fig. 1(a). The pattern of matrix ($N = 100$) in Test 1 by using rowwise ordering. Dots indicate the nonzeros.

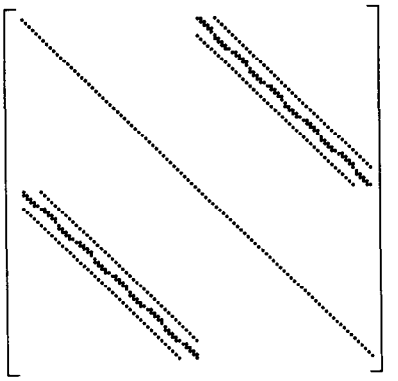


Fig. 1(b). The pattern of matrix ($N = 100$) in Test 1 by using Red/Black ordering. Dots indicate the nonzeros.

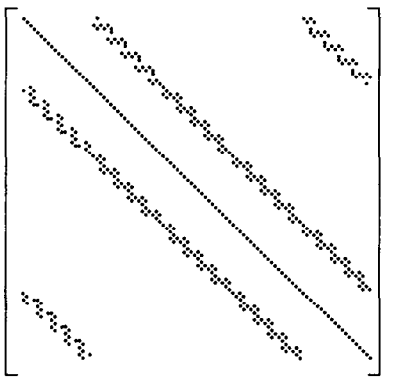


Fig. 1(c). The pattern of matrix ($N = 100$) in Test 1 by using 3-colour ordering. Dots indicate the nonzeros.

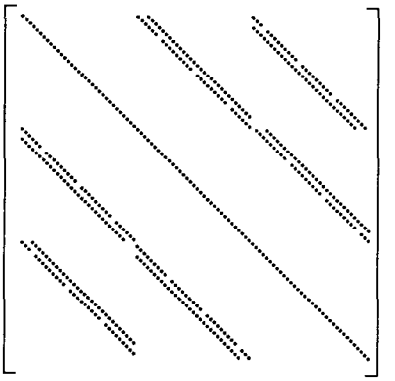


Fig. 1(d). The pattern of matrix ($N = 100$) in Test 1 by using 5-colour ordering. Dots indicate the nonzeros.

We choose $n = 10$, 20, 30, and the corresponding ω for each kind of ordering is shown in Table 1. $N = n \times n$ is the number of unknowns and n is the number of grid points along every line of the mesh.

In Table 1 for the natural rowwise ordering, we choose ω_{opt} by the formula

$$\omega_{opt} = \frac{1}{1 + \sqrt{1 - \rho^2}}, \quad \rho = \cos\left(\frac{\pi}{n}\right).$$

For the 2-, 3- and 5-colour orderings, we choose ω_{opt} by testing several times. We start the test

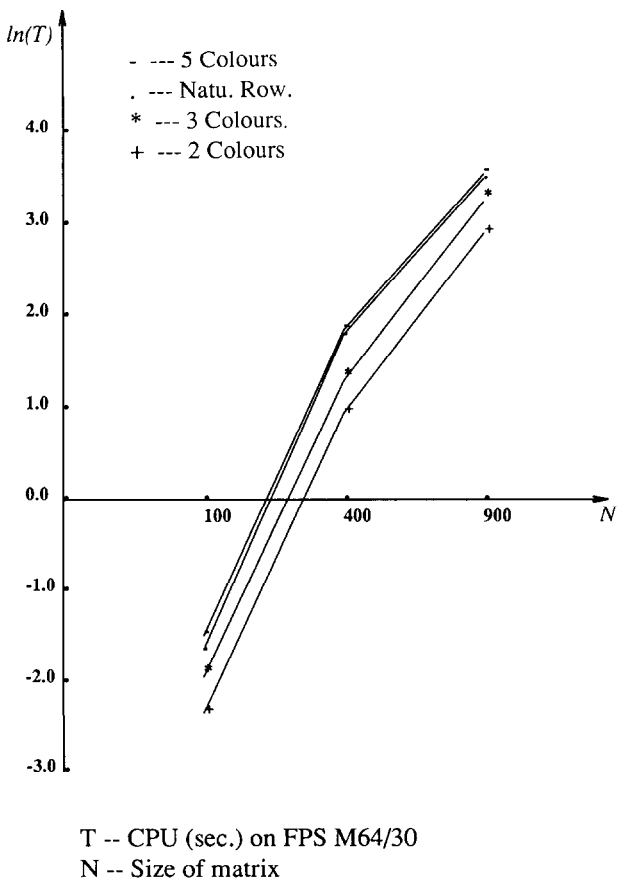
Table 1
The optimal ω_{opt} for Test 1

N	Natural rowwise	2 colours	3 colours	5 colours
100	1.6010336	1.6195289	1.6195228	1.6195228
400	1.8023706	1.8036030	1.8003910	1.8003910
900	1.8231664	1.8208600	1.8220200	1.8220200

Table 2
Result of Test 1

	Kind of ordering	Iteration number	CPU [a] of SOR (sec.)	CPU [b] of colouring (sec.)
$n=10$	Natural rowwise	29	0.1860	—
	2 colours	28	$0.9926 \cdot 10^{-1}$	$0.5037 \cdot 10^{-2}$
	3 colours	29	0.1435	$0.9726 \cdot 10^{-2}$
	5 colours	29	0.2317	$0.1181 \cdot 10^{-1}$
$n=20$	Natural rowwise	63	5.8165	—
	2 colours	59	2.6637	$0.1625 \cdot 10^{-1}$
	3 colours	60	3.6637	$0.3436 \cdot 10^{-1}$
	5 colours	60	5.8628	$0.4228 \cdot 10^{-1}$
$n=30$	Natural rowwise	71	32.502	—
	2 colours	67	18.239	$0.3499 \cdot 10^{-1}$
	3 colours	66	25.214	$0.7672 \cdot 10^{-1}$
	5 colours	71	33.014	$0.9330 \cdot 10^{-1}$

[a] The CPU-time for SOR iteration only.
[b] The CPU-time of reordering matrix by the colouring algorithm described in the previous section. For the natural rowwise ordering there is no such reordering needed.



T -- CPU (sec.) on FPS M64/30
N -- Size of matrix

Fig. 2(a). The logarithm of CPU as a function of the size of the matrix.

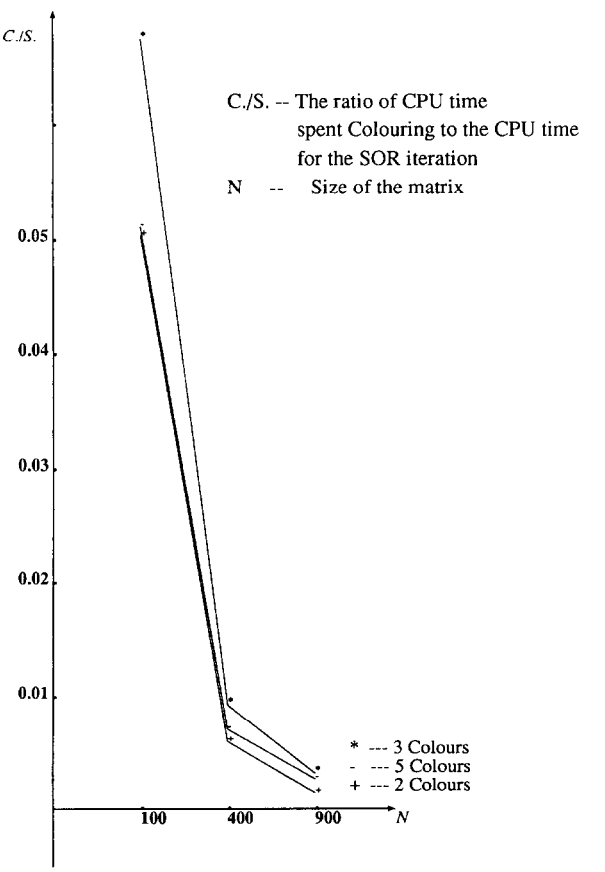


Fig. 2(b). The ratio of the CPU-time spent colouring to the CPU-time for the SOR iteration as a function of the size of the matrix.

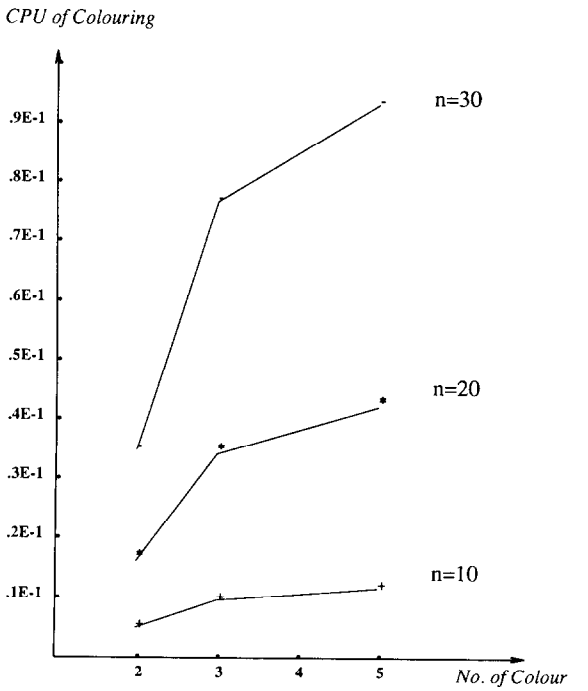


Fig. 2(c). The CPU-time spent colouring as a function of the number of colours.

from the corresponding values for the natural rowwise ordering. We increase and decrease the value to find an optimal one. This method is just used for testing. More practical methods can be found in Reid [10].

The number of iterations, the CPU-time of the SOR method or the multicolour SOR method and the CPU-time of colouring for the last three kinds of ordering are shown in Table 2. In this table, "Natural rowwise" and "2 colours" etc. indicate the kind of ordering of the unknowns, n is the number of grid points along each line of the mesh.

We compare the results in the above three figures. In Fig. 2(a), we compare the CPU-time of the four different kinds of ordering. Since the FPS M64/30 is a pipelined computer, the Red/Black ordering is the best one, this was analysed in the previous section. When the number of colours increases, the CPU-time of the multicolour SOR method increases. For a pipelined computer the longer the vector to be computed in parallel, the more efficient the algorithm will be. The more colour groups there are, the shorter the vector is, i.e., the less efficient the multicolour SOR is. This explains why the 5-colour ordering is worse than the natural rowwise ordering on a pipelined computer, though in theory the coloured ordering has more parallel property than the natural rowwise ordering does. But for a parallel computer we expect the 3-colour or the 5-colour ordering to be better than the natural rowwise ordering. Even for a pipelined computer, if we increase n, we can expect that the 5-colour ordering uses less CPU-time than the natural rowwise ordering. Since from the results in Table 1 we can see that the difference of the CPU-time for the 5-colour ordering and for the natural rowwise ordering per iteration (we denote it as p) decreases when n increases, when $n = 10$, $p = 0.0846$, when $n = 20$, $p = 0.000925$ and when $n = 30$, $p = 0.00022$.

We use C/S to show the ratio of the CPU-time spent colouring to the CPU-time for all the SOR iterations. In Fig. 2(b), for different n, i.e., a different linear system, we showed C/S. We discover that the bigger the linear system is, the less the C/S is. In Fig. 2(c), we show the CPU-time of colouring for different linear systems and different numbers of colour. We can see that the bigger the linear system is, the longer the CPU-time of colouring is. We also find that the more colours we use, the longer the CPU-time is.

From this test, we show the basic properties of the multicolour SOR method and the colouring algorithms. Now we use these methods to solve a linear system obtained from the finite-element discretisation.

Test 2. We use the test problem in [8]. That is

$$u_{xx}(x, y) + 2u_{yy}(x, y) = 0, \quad (x, y) \in S = (0, 1) \times (0, 1),$$
$$u(x, y) = 1 + xy, \quad (x, y) \in \text{boundary of } S.$$

We choose a triangular mesh in S. The grid points are placed randomly in the interior of S and are controlled at a distance from the boundary. On the boundary the grid points are placed uniformly, there are 21 points on each edge of the boundary.

We use the linear triangle element to discretise the partial differential equation. There are 441 grid points in the mesh. When the mesh is produced the grid points have an order which we call natural ordering. To order the grid points, we need 5 colours using Algorithm C1 (we call the ordering of the unknowns C1 ordering) and 9 colours using Algorithm C2 (we call the ordering of the unknowns C2 ordering).

The ω_{opt} we use for each ordering is shown in Table 3, which are obtained similar to those in Test 1.

Similar to Table 2, in Table 4 we present the number of iterations, the CPU-time of the SOR method or the multicolour SOR method and the CPU-time of colouring for the last two kinds of ordering.

Table 3
The optimal ω_{opt} for Test 2

Natural ordering	C1 ordering	C2 ordering
1.5432915	1.5479724	1.5338425

Table 4
Result of Test 2

Kind of ordering	Iteration number	CPU [a] of SOR (sec.)	CPU [b] of colouring (sec.)
Natural ordering	60	6.7475	—
C1 ordering	56	5.5617	$0.2589 \cdot 10^{-1}$
C2 ordering	57	7.2349	0.7403

[a] The CPU-time for SOR iteration only.
[b] The CPU-time of reordering matrix by the colouring algorithm described in the previous section. For the natural ordering there is no such reordering needed.

Table 5
CPU (sec.) for each group of node

Group number	CPU (sec.) of SOR [a]	Number of node in each group	CPU (sec.) per node [b]
1	1.4055	121	0.0116
2	1.3232	109	0.0121
3	1.2692	102	0.0124
4	1.2472	99	0.0126
5	0.2733	10	0.0273

[a] The CPU-time for SOR iteration for each group of node.
[b] The average CPU-time of SOR iteration for each node in each group.

Like Test 1 the C1 ordering is the best one. But as explained above, the multicolour SOR method is not efficient enough since too many colour groups are involved. So the speedup of the multicolour SOR method for the C1 ordering to the natural SOR method is not very high. And the multicolour SOR method for the ordering obtained by the C2 algorithm is not suitable for pipelined computers. In order to show the property of a pipelined computer, we present the CPU-time of the multicolour SOR method for each group of the C1 ordering, the number of grid points in each group and the CPU-time per node in each group. (Denoted as "CPU per node" in Table 5.)

From Table 5 we can see that the more grid points there are in each group, the less the CPU-time is per node, i.e., the longer the vector is, the more efficient the multicolour SOR method is. This is just the property of a pipelined computer.

4. Conclusion

The Red/Black ordering can be generalised to a general mesh for the finite-element method. For the finite-element method, before we begin the real process we can give a suitable ordering of the mesh points and a suitable ordering of the elements for the assembly by using the algorithms given in this paper which are dependent on the mesh. The multicolour SOR method for the finite-element method has the same properties as the Red/Black SOR for the finite-difference method on a regular stencil when it is used on both vector and parallel computers.

When we find a reordering algorithm we should pay more attention to the finite-element mesh and the properties of different kinds of computers, as these may be different for both vector and parallel computers.

Acknowledgements

The author wishes to thank all the members of the NAG in T.C.D. for their helpful discussions and for supplying the necessary data (such as the mesh for Test 2). I am especially grateful to Professor J.J.H. Miller and Dr. C.J. Fitzsimons for encouraging me to study this interesting problem.

References

[1] L.M. Adams and H.F. Jordan, Is SOR color-blind?, *SIAM J. Sci. Statist. Comput.* **7** (1986) 490–506.
[2] L.M. Adams, R.J. LeVegue and D.M. Young, Analysis of the SOR iteration for the 9-point Laplacian, *SIAM J. Numer. Anal.* **25** (1988) 1156–1180.
[3] L.M. Adams and J. Ortega, A multi-color SOR method for parallel computation, in: *Proc. 1982 Internat. Conf. on Parallel Processing*, Bellaire, MI, 1982, 53–58.
[4] R.E. Bank, W.M. Coughran Jr., W. Fichtner, D.J. Rose and R.K. Smith, Computational aspects of semiconductor device simulation, in: W.L. Engl, Ed., *Process and Device Modeling* (North-Holland, Amsterdam, 1986).
[5] Ph. Berger, P. Brouays and J.C. Syre, A mesh coloring method for efficient MIMD processing in finite element problems, in: *Proc. 1982 Internat. Conf. on Parallel Processing*, Bellaire, MI, 1982, 41–52.
[6] A. George, Nested dissection of a regular finite element mesh, *SIAM J. Numer. Anal.* **10** (1973) 345–362.
[7] L. Hayes, Comparative analysis of iterative techniques for solving Laplace's equation on the unit square on a parallel processor, M. Sc. Thesis, Dept. Math., Univ. Texas, Austin, 1974.
[8] D.R. Kincaid, T.C. Oppe and D.M. Young, Vector computations for sparse linear systems, *SIAM J. Algebraic Discrete Methods* **7** (1986) 99–112.
[9] J. Lambiotte, The solution of linear systems of equations on a vector computer, Ph.D. Dissertation, Univ. Virginia, 1975.
[10] J.K. Reid, A method for finding the optimum successive over-relaxation parameter, *Comput. J.* **9** (1966) 200–204.
[11] R.S. Varga, *Matrix Iterative Analysis* (Prentice-Hall, Englewood Cliffs, NJ, 1962).